Implementing zFilter based forwarding node on a NetFPGA

Jari Keinänen, Petri Jokela, Kristian Slavov
Ericsson Research, NomadicLab
02420 Jorvas, Finland
firstname.secondname@ericsson.com

ABSTRACT

Our previous work has produced a novel, Bloom-filter based, forwarding fabric, suitable for large-scale topic-based publish/subscribe [8]. Due to very simple forwarding decisions and small forwarding tables, the fabric may be more efficient than the currently used ones. In this paper, we describe the NetFPGA based forwarding node implementation for this new, IP-less, forwarding fabric. The implementation requires removing the traditional IP forwarding implementation, and replacing it with the Bloom-filter matching techniques for making the forwarding decisions. To complete the work, we provide measurement results to verify the forwarding efficiency of the proposed forwarding system and we compare these results to the measurements from the original, IP-based forwarding, implementation.

1. INTRODUCTION

While network-level IP multicast was proposed almost two decades ago [5], its success has been limited due to the lack of wide scale deployment. As a consequence, various forms of application-level multicast have gained in popularity, but their scalability and efficiency have been limited. Hence, a challenge is how to build a multicast infrastructure that can scale to, and tolerate the failure modes of, the general Internet, while achieving low latency and efficient use of resources.

In [8], we propose a novel multicast forwarding fabric. The mechanism is based on identifying links instead of nodes and uses in-packet Bloom filters [2] to encode source-route-style forwarding information in the packet header. The forwarding decisions are simple and the forwarding tables fairly small, potentially allowing faster, smaller, and more energy-efficient switches than what today's switches are. The proposed (inter-)networking model aims towards balancing the state between the packet headers and the network nodes, allowing both stateless and stateful operations [16].

The presented method takes advantage of "inverting" the Bloom filter thinking [3]. Instead of maintaining Bloom filters at the network nodes and verifying from incoming packets if they are included in the filter or not, we put the Bloom filters themselves in the packets and allow the nodes on the path to determine which outgoing links the packet should be forwarded to.

In this paper, we present the implementation of a forwarding node on a NetFPGA. At the first stage, we have implemented the basic forwarding node functions enabling packet delivery through the network using the described forwarding mechanism. At the same time we have been developing a FreeBSD-based end-host implementation, based on publish/subscribe networking architecture, described in [8]. The end-host implements the packet management, as well as networking related functions. The present environment supports only simple networks, but once the first release of the end-host implementation is ready, larger scale networks can be created and tested.

We selected NetFPGA as the forwarding node platform because it offers a fast way to develop custom routers. It provides a way easy to move implementations directly on hardware by taking advantage of reprogrammable FPGA circuits enabling prototype implementations that can handle high speed data transmission (1Gbps/link). We can also avoid time consuming and expensive process of designing new physical hardware components.

The rest of this paper is organized as follows. First, in Section 2, we discuss the general concepts and architecture of our solution. In Section 3, we go into details of the implementation. Next, in Section 4, we provide some evaluation and analysis of our forwarding fabric Section 5 contrasts our work with related work, and Section 6 concludes the paper.

2. ARCHITECTURE

Our main focus in this paper is on describing the forwarding node implementation of the Bloom-filter based forwarding mechanism referred to as zFilters. In this section, we describe the basic zFilter operations, and for more detailed description, we refer to [8].

2.1 Forwarding on Bloomed link identifiers

The forwarding mechanism described in this paper

is based on identifying links instead of nodes. In the basic operation, the forwarding nodes do not need to maintain any state other than a Link ID per interface. The forwarding information is constructed using these Link IDs and including them in the packet header in a Bloom filter fashion. For better scalability, we introduce an enhancement that inserts a small amount of state in the network by creating virtual trees in the network and identifying them using similar identifiers as the Link IDs. In this section we describe the basics of such forwarding system, and more detailed information can be found from [8].

2.1.1 The basic Bloom-filter-based forwarding

For each point-to-point link, we assign two identifiers, called Link IDs, one in each direction. For example, a link between the nodes A and B has two identifiers, $\overrightarrow{AB}$ and $\overleftarrow{AB}$. In the case of a multi-point link, such as a wireless link, we consider each pair of nodes as a separate link. With this setup, we don't need any common agreement between the nodes on the link identities – each link identity may be locally assigned, as long as the probability of duplicates is low enough.

Basically, a Link ID is an m-bit long name with just k bits set to one. In [8] we discuss the proper values for m and k, and what are the consequences if we change the values; however, for now it is sufficient to note that typically $k \ll m$ and m is relatively large, making the Link IDs statistically unique (e.g., with $m = 248$, $k = 5$, # of Link IDs $\approx m!/(m-k)! \approx 9 * 10^{11}$).

The complete architecture includes a management system that creates a graph of the network using Link IDs and connectivity information, without any dependency on end-point naming or addressing (creating the "topology map" or "routing table"). Using the network graph, the topology system can determine a forwarding tree for any publication, from the locations of the publisher and subscribers [16]. In this paper, however, we assume that such topology management exists and refer to [8] for more detailed discussion about the complete architecture.

When the topology system gets a request to determine a forwarding tree for a certain publication, it first creates a conceptual delivery tree for the publication using the network graph. Once it has such an internal representation of the tree, it knows which links the packets need to pass, and it can determine when to use Bloom filters and when to create state. [16]

In the default case, we use a source-routing based approach which makes forwarding independent from routing. Basically, we encode all Link IDs of the delivery tree into a Bloom filter, forming the forwarding zFilter for the data. Once all Link IDs have been added to the filter, a mapping from the data topic identifier to the zFilter is given to the node acting as the data source, which now can create packets that will be delivered along the tree.

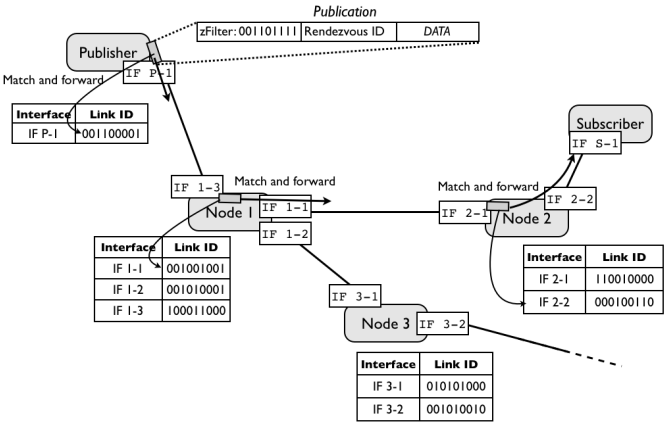

Figure 1: Example of Link IDs assigned for links, as well as a publication with a zFilter, built for forwarding the packet from the Publisher to the Subscriber.

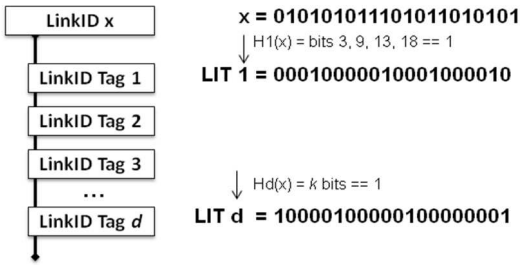

Figure 2: An example relation of one Link ID to the d LITs, using k hashes on the Link ID.

Each forwarding node acts on packets roughly as follows. For each link, the outgoing Link ID is ANDed with the zFilter found in the packet. If the result matches with the Link ID, it is assumed that the Link ID has been added to the zFilter and that the packet needs to be forwarded along that link.

With Bloom filters, matching may result with some false positives. In such a case, the packet is forwarded along a link that was not added to the zFilter, causing extra traffic. While the ratio of false positives depends on the number of entries added to the filter, we get a practical limit on how many link names can be included into a single zFilter.

Our approach to the Bloom filter capacity limit is twofold: Firstly, we use recursive layering [4] to divide the network into suitably-sized components and secondly, the topology system may dynamically add *virtual links* to the system (see Section 2.2.1).

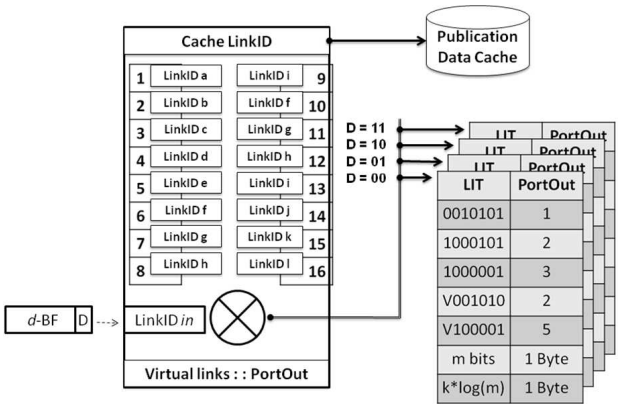

Figure 3: Outgoing interfaces are equipped with d forwarding tables, indexed by the value in the incoming packet.

2.1.2 Link IDs and LITs

To reduce the number of false positives, we introduced [8] *Link ID Tags* (LITs), as an addition to the plain Link IDs. The idea is that instead of each link being identified with a single Link ID, every unidirectional link is associated with a set of d distinct LITs (Fig. 2). This allows us to construct zFilters that can be optimized, e.g., in terms of the false positive rate, compliance with network policies, or multi path selection. The approach allows us to construct different candidate zFilters and to to select the best-performing Bloom filter from the candidates, according to any appropriate metric.

The forwarding information is stored in the form of d forwarding tables, each containing the LIT entries of the active Link IDs, as depicted in Fig. 3. The only modification of the base forwarding method is that the node needs to be able to determine which forwarding table it should perform the matching operations; for this, we include the index in the packet header before zFilter.

The construction of the forwarding Bloom filter is similar to the one discussed in single Link ID case, except that for the selected path from the publisher to the subscriber, we calculate d candidate filters, one using each of the d values, which are each equivalent representations of the delivery tree.

As a consequence, having d different candidates each representing the given delivery tree is a way to minimise the number of false forwardings in the network, as well as restricting these events to places where their negative effects are smallest. [8]

2.2 Stateful operations

In the previous, we presented the basic, single link, based forwarding solution. A forwarding node does not maintain any connection or tree based states, the only information that it has to maintain is the outgoing Link IDs. In this section, we discuss some issues that enhance the operation with the cost of adding small amount of state on the forwarding nodes.

2.2.1 Virtual links

As discussed in [8], the forwarding system in its basic form, is scalable into metropolitan area networks with sparse multicast trees. However, in case of dense multicast trees and larger networks, increasing the number of Link IDs in the Bloom filter will increase the number of false positives on the path. For more efficient operations, the topology layer can identify different kinds of delivery trees in the network and assign them virtual Link IDs that look similar to Link IDs described earlier.

Once a virtual link has been created, each participating router is configured with the newly created virtual Link ID information, adding a small amount of state in its forwarding table. The virtual link identifier can then be used to replace all the single Link IDs needed to form the delivery tree, when creating zFilters.

2.2.2 Link failures - fast recovery

All source routing based forwarding mechanisms are vulnerable when link failures occur in the network. While the packet header contains the exact route, the packets will not be re-routed using other paths.

In zFilters [8], we have proposed two simple solutions for this mentioned problem: we can use either pre-configured virtual links, having the same Link ID as the path which it is replacing or then we can use pre-computed zFilters, bypassing the broken link. The former method requires an additional signalling message so that the alternative path is activated, but the data packets can still use the same zFilter and do not need any modifications. The latter solutions requires that the alternative path is added to the zFilter in the packet header, thus increasing the fill factor of the zFilter, increasing the probability of false positives. However, the solution does not require any signalling when the new path is needed.

2.2.3 Loop prevention

The possibility for false positives means that there is a risk for loops in the network. The loop avoidance has also been discussed in [8] with some initial mechanisms for avoiding such loops. Locally, it is possible to calculate zFilter that do not contain loops, but when the packet is passed to another administrative domain, it is not necessarily possible. One other alternative is to use TTL-like field in the packet for removing looping packets. Work is going on in this area.

2.3 Control messages, slow path, and services

To inject packets to the slow path on forwarding nodes, each node can be equipped with a local, unique Link ID denoting the node-internal passway from the switching fabric to the control processor. That allows targeted control messages that are passed only to one or a few nodes. Additionally, there may a be virtual Link ID attached to these node-local passways, making it possible to multicast control messages to a number of forwarding nodes without needing to explicitly name each of them.

By default such control messages would be simultaneously passed to the slow path and forwarded to the neighboring nodes. The simultaneous forwarding can be blocked easily, either by using zFilters constructed for node-to-node communication, or using a virtual Link ID that is both configured to pass messages to the slow path and to block them at all the outgoing links.

Generalising, we make the observation that the egress points of a virtual link can be basically anything: nodes, processor cards within nodes, or even specific services. This allows our approach to be extended to upper layers, beyond forwarding, if so desired.

3. IMPLEMENTATION

In the project, we have designed a publish/subscribe based networking architecture with a novel forwarding mechanism. The motivation to choose NetFPGA as the platform for our forwarding node implementation was based on our requirements. We needed a platform that was capable for high data rates and has the flexibility that allows implementation of a completely new forwarding functionality.

The current implementation has roughly 500 lines of Verilog code, and it implements most of the functions described in the previous section. In this section, we will go deeper in the implementation and describe what changes we have made to the original reference implementation.

3.1 Basic forwarding method

Algorithm 1: Forwarding method of LIPSIN

Input: Link IDs of the outgoing links; zFilter in the packet header

```
foreach Link ID of outgoing interface do
    if zFilter & Link ID == Link ID then
        Forward packet on the link
    end
end
```

The core operation of our forwarding node is to make the forwarding decision for incoming packets. With zFilters, the decision is based on a binary AND and comparison operations, both of which are very simple to implement in hardware. The forwarding decision

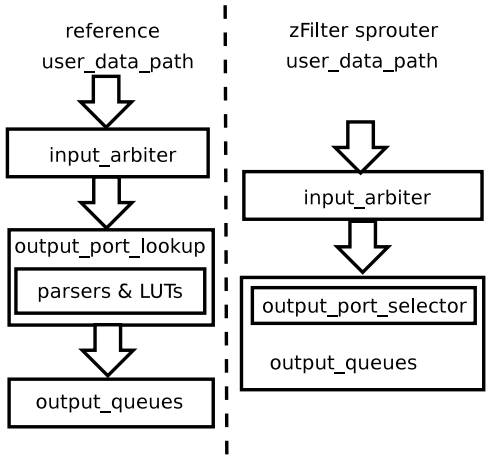

Figure 4: Reference and modified datapaths

(Alg. 1) can be easily parallelized, as there are no memory or other shared resource bottlenecks. The rest of this section describes the implementation based on this simple forwarding operation.

3.2 Forwarding node

For the implementation work we identified all unnecessary parts from the reference switch implementation and removed most of the code that is not required in our system (Figure 4). The removed parts were replaced with a simple zFilter switch.

The current version implements both the LIT and the virtual link extensions, and it has been tested with four real and four virtual LITs per each of the four interface. We are using our own *EtherType* for identifying zFilter packets. The implementation drops incoming packets with wrong ethertype, invalid zFilter, or if the TTL value has decreased down to zero.

The *output_port_lookup* module and all modules related to that are removed from the reference switch design. The zFilter implementation is not using any functions from those modules.

Our prototype has been implemented mainly in the new *output_port_selector* module. This module is responsible for the zFilter matching operations, including binary AND operation between the LIT and zFilter, and comparing the result with the LIT, as well as placing the packets to the correct output queues based on the matching result. The new module is added in *output_queueus*. Detailed structure of the *output_port_selector* module is shown in Figure 5.

3.2.1 Packet forwarding operations

All incoming data is forwarded, straight from the input arbiter, to the *store_packet* module, that stores it into the SRAM and to the *output_port_selector* mod-

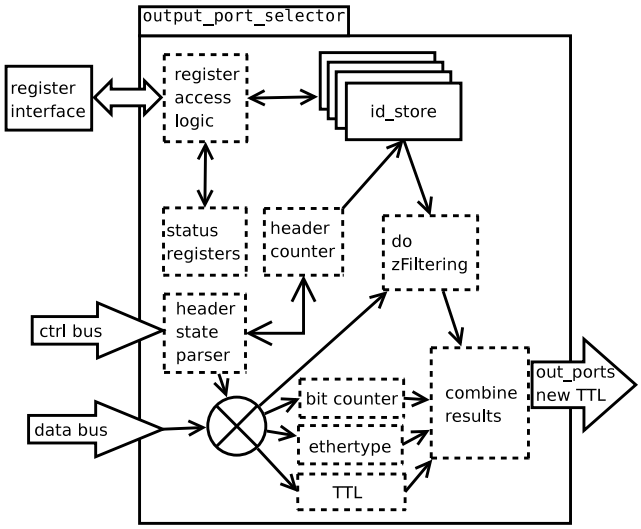

Figure 5: Structure of the output port selector module

ule for processing. Packets arrive in 64-bit pieces, one piece on each clock cycle. The packet handling starts with initiating processing for different verifications on the packet as well as on the actual zFilter matching operation.

The incoming packet processing takes place in various functions, where different kinds of verifications are performed to the packet. The three parallelized verification operations, *bit_counter*, *ethertype*, and *TTL*, make sanity checks on the packet, while the *do zFiltering* makes the actual forwarding decisions. In practice, for enabling parallelization, there exists separate instances of logic blocks that do zFiltering, one for each of the Link IDs (both for ordinary and virtual links). In the following, we go through the functions in Figure 5 function-by-function and in Figure 6, the operations are shown in function of clock cycles.

In *do zFiltering*, we make the actual zFilter matching for each 64-bit chunk. First, we select the correct LITs of each of the interfaces based on the d-value in the incoming zFilter. For maintaining the forwarding decision status for each of the interfaces during the matching process, we have a bit-vector where each of the interfaces has a single bit assigned, indicating the final forwarding decision. Prior to matching process, all the bits are set to one. During the zFilter matching, when the system notices that there is a mismatch in the comparison between the AND-operation result and the LIT, the corresponding interface's bit in the bit-vector is set to zero.

Finally, when the whole zFilter has been matched with the corresponding LITs, we know the interfaces where the packet should be forwarded by checking from the bit-vector, which of the bits are still ones. While the forwarding decision is also based on the other verifications on the packet, the *combine results* collects the information from the three verification functions in addition to the zFilter matching results. If all the collected verification function results indicate positive forwarding decision, the packet will be put to all outgoing queues indicated by the bit vector. The detailed operations of the verification functions are described in 3.2.2.

3.2.2 Support blocks and operations

To avoid the obvious attack of setting all bits to one in the zFilter, and delivering the packet to all possible nodes in the network, we have implemented a very simple verification on the zFilter. We have limited the maximum number of bits set to one in a zFilter to a constant value, which is configurable from the user space; if there are more bits set to one than the set maximum value, the packet is dropped. The bit counting function calculates the number of ones in a single zFilter and it is implemented in the *bit_counter* module. This module takes 64 bits wide input and it returns the amount of ones on the given input. Only wires and logic elements are used to calculate the result and there are no registers inside, meaning that block initiating the operation should take care of the needed synchronization.

The *Ethertype* of the packet is checked upon arrival. At the moment, we are using 0xacdc as the ethertype, identifying the zFilter-based packets. However, in a pure zFilter based network, the ethernet is not necessarily needed, thus this operation will be obsolete. The third packet checking operation is the verification of the TTL. This is used in the current implementation to avoid loops in the network. This is not an optimal solution for loop prevention, and better solutions are currently being worked on.

The *id_store* module implements Dual-Port RAM functionality making it possible for two processes to access it simultaneously. This allows modifications to LIT:s without blocking forwarding functionality. The *id_store* module is written so that it can be synthesized by using either logic cells or BRAM (Block RAM). One of the ports is 64 bits wide with only read access and it is used exclusively to get IDs for zFiltering logic. There is one instance of the *id_store* module for each LIT and virtual LIT. This way the memory is distributed and each instance of the filtering logic have access to *id_store* at line rate. The other port of the *id_store* module is 32 bits wide with a read and write connection for the user space access. This port is used by the management software (cf. Section 3.2.3) to configure LIT:s on the interfaces. One new register block is reserved for this access.

One additional register block is added for module control and debug purposes. It is used to read information that is collected into the status registers during forwarding operations. Status registers contain constants,

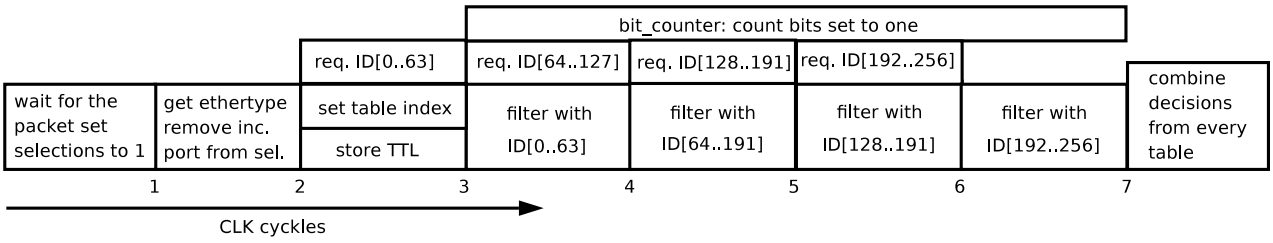

Figure 6: Dataflow diagram

amount of links, maximum amount of LITs and virtual LITs per link and also the LIT length. In addition, information about the last forwarded packet is stored together with the result of the bit count operation, d, TTL, and incoming port information. This block is also used to set the maximum amount of ones allowed in a valid zFilter.

3.2.3 *Management software*

For configuration and testing purposes, we have developed a specialized management software. When the system is started, the management software is used to retrieve information from the card and, if needed, to configure new values on the card. The information that the software can handle, includes the length of the LITs, the maximum d value describing the number of LITs used, as well as both link and virtual link information from each of the interfaces.

Internally, the software works by creating chains of commands that it sends in batch to the hardware, gets the result and processes the received information. The commands are parsed using specific, for the purpose generated, grammar. The parsing is done using byacc and flex tools, and is therefore easily extendable.

For testing purposes, the software can be instructed to send customizable packets to the NetFPGA card, and to collect information about the made forwarding decisions. The software supports the following features:

- Selecting the outgoing interface
- Customizing the delay between transmitted packets
- Varying the sizes of packets
- Defining the Time-to-live (TTL) field in packet header
- Defining the d value in packet header
- Defining the zFilter in the packet header
- Defining the ethernet protocol field

# of NetFPGAs	Average latency	Std. Dev.	Latency/ NetFPGA
0	$16\mu s$	$1\mu s$	N/A
1	$19\mu s$	$2\mu s$	$3\mu s$
2	$21\mu s$	$2\mu s$	$3\mu s$
3	$24\mu s$	$2\mu s$	$3\mu s$

Table 1: Simple latency measurement results

4. EVALUATION

The basic functionality is tested by running simple scripts that use control software (cf. Section 3.2.3) to set Link IDs and to generate and send traffic. In practise, two network interfaces of the test host are connected to the NetFPGA of which one is used to send packets to the NetFPGA and the other one to receive forwarded packets. Forwarding decisions are also followed by tracking status registers. The results of the tests show that the basic forwarding functions work on the NetFPGA, also when using LITs. In addition, the packet verification operations, counting set bits, TTL verification, as well as ethertype checking were working as expected.

4.1 Performance

To get some understanding of the potential speed, we measured packet traversal times in our test environment. The first set of measurements, shown in Table 1, focused on the latency of the forwarding node with a very low load. For measurements, we had four different setups, with zero (direct wire) to three NetFPGAs on the path. Packets were sent at the rate of 25 packets/second; both sending and receiving operations were implemented directly in FreeBSD kernel.

The delay caused by the Bloom filter matching code is $64ns$ (8 clock cycles), which is insignificant compared to the measured $3\mu s$ delay of the whole NetFPGA processing. With background traffic, the average latency per NetFPGA was increased to $5\mu s$.

To get some practical reference, we also compared our implementation with the Stanford reference router. This was quantified by comparing ICMP echo requests' processing times with three setups: using a plain wire,

Path	Avg. latency	Std. Dev.
Plain wire	$94\mu s$	$28\mu s$
IP router	$102\mu s$	$44\mu s$
LIPSIN	$96\mu s$	$28\mu s$

Table 2: Ping through various implementations

using our implementation, and using the reference IP router with five entries in the forwarding table. To compensate the quite high deviation, caused by sending and receiving ICMP packets and involving user level processing, we averaged over 100 000 samples. Both IP router implementation and our implementation were run on the same NetFPGA hardware. The results are shown in Table 2.

While we did not directly measure the bandwidth due to the lack of test equipment for reliably filling up the pipes, there are no reasons why the implementation would not operate at full bandwidth. To further test this we did send video stream through our implmentation. During the streaming we did send random data through same NetFPGA but with different ports at almost 1Gbs datarate. Both the stream and was forwarded without a problem. The code is straightforward and should be able to keep the pipeline full under all conditions.

IP routers need increasing amount of states, which increases latency and resource consumption, when the size of the network increases. On the other hand, in our implementation, the latency and resource consumption for each node will remain same, independent of the amount of nodes in the network. Because of that, the results we got for one node should remain same even when large amount of nodes are connected to a same network.

4.2 Resource consumption:

To get an idea how much our implementation consumes resources we did syntetize design with 4 real and 4 virtual LITs per interface. With this configuration, the total usage of NetFPGA resources for the forwarding logic is 4.891 4-input LUTs out of 47.232, and 1.861 Slice Flip/Flops (FF) out of 47.232. No BRAMs are reserved for the forwarding logic. Synthetisizer saves BRAM blocks and uses other logic blocks to create registers for LITs. For the whole system, the corresponding numbers are 20.273 LUTs, 15.347 FFs, and 106 BRAMs. SRAM was used for the output queues in the measured design. We also tested to use BRAMs for output queue and the design works. However, we don't have measurement results from that implementation.

4.3 Forwarding table sizes:

Assuming that each forwarding node maintains d distinct forwarding tables, each containing an entry per interface, where an entry further consists of a Link ID and the associated output port, we can estimate the amount of memory needed by the forwarding tables:

$$FT_{mem} = d \cdot \#Links \cdot [size(LIT) + size(P_{out})] \quad (1)$$

Considering $d = 8$, 128 links (physical & virtual), 248-bit LITs and 8 bits for the outport, the total memory required would be $256Kbit$, which easily fits on-chip.

Although this memory size is already small, we can design an even more efficient forwarding table by using a *sparse representation* to store just the positions of the bits set to 1. Thereby, the size of each LIT entry is reduced to $k \cdot log_2(LIT)$ and the total forwarding table requires only $\approx 48Kbit$ of memory, at the expense of the decoding logic.

5. RELATED WORK

OpenFlow [11] [12] provides a platform for experimental switches. It introduces simple, remote controlled, flow based switches, that can be run on existing IP switches. The concept allows evaluation of new ideas and even protocols in Openflow-enabled networks, where the new protocols can be run on top of the IP network. However, as high efficiency is one of our main goals, we wanted to get rid of unnecessary logic and decided for a native zFilter implementation.

There are not yet many publications where NetFPGA is used, in addition to OpenFlow and publications about implementing the NetFPGA card or reference designs. However one technical report were available [10], about implementing flow counter on NetFPGA. Authors of that work used NetFPGA succesfully to demonstrate that their idea can be implemented in practice.

In addition to NetFPGA, there are also other reconfigurable networking hardware approaches. For instance, [9] describes one alternative platform. There are also other platforms that could work for this type of development, for example Combo cards from Liberouter project [1]. However, NetFPGA provides enough speed and resources for our purposes, but Combo cards might become a good option later on if we need higher line speeds.

In the following we briefly discuss some work in the area of forwarding related to our zFilter proposal.

IP multicast: Our basic communication scheme is functionally similar to IP-based source specific multicast (SSM) [6], with the IP multicast groups having been replaced by the topic identifiers. The main difference is that we support stateless multicast for sparse subscriber groups, with unicast being a special case of multicast. On the contrary, IP multicast typically creates lots of state in the network if one needs to support a large set of small multicast groups.

Networking applications of Bloom filters:

For locating named resources, BFs have been used to bias random walks in P2P networks [3]. In content-based pub/sub systems [7], summarized subscriptions are created using BFs and used for event routing purposes. Bloom filters in packet headers were proposed in Icarus [14] to detect routing loops, in [15] for credentials-based data path authentication, and in [13] to represent AS-level paths of multicast packets in a 800-bit shim header, `TREE_BF`. Moreover, the authors of [13] use Bloom filters also to aggregate active multicast groups inside a domain and compactly piggyback this information in BGP updates.

6. CONCLUSIONS

Previously, we have proposed a new forwarding fabric for multicast traffic. The idea was based on reversing Bloom filter thinking and placing a Bloom filter into the delivered data packets. Our analysis showed that with reasonably small headers, comparable to those of IPv6, we can handle the large majority of Zipf-distributed multicast groups, up to some 20 subscribers, in realistic metropolitan-sized topologies, without adding any state in the network and with negligible forwarding overhead. For the remainder of traffic, the approach provides the ability to balance between stateless multiple sending and stateful approaches. With the stateful approach, we can handle dense multicast groups with a very good forwarding efficiency. The forwarding decisions are simple, potentially energy efficient, may be parallelized in hardware, and have appealing security properties.

To validate and test those claims, we implemented a prototype of a forwarding node and tested its performance. As described in Chapter 4, we ran some measurements on the forwarding node and concluded that the whole NetFPGA processing for a zFilter creates a $3\mu s$ delay. This delay could most likely be reduced, because the forwarding operation should take only $64ns$ (8 clock cycles). Comparison with the IP router implementation was done by using ICMP echo requests, showing zFilter implementation being slightly faster than IP-based forwarding, running on the same NetFPGA platform.

Our simple implementation still lacks some of the advanced features described in [8], for example reverse path creation and signaling. However, it should be quite straightforward to add those features to the existing design. Also, early studies indicate that it should be possible to start adding even more advanced features, like caching, error correction or congestion control to the implementation.

7. REFERENCES

[1] Liberouter. http://www.liberouter.org/.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of Bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2004.

[4] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.

[5] S. E. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. on Comp. Syst.*, 8(2), 1990.

[6] H. Holbrook and B. Cain. Source-specific multicast for IP. RFC 4607. Aug 2006.

[7] Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS '08*, pages 71–81, New York, NY, USA, 2008. ACM.

[8] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander. LIPSIN: Line speed publish/subscribe inter-networking. Technical report, www.psirp.org, 2009.

[9] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2001.

[10] J. Luo, Y. Lu, and B. Prabhakar. Prototyping counter braids on netfpga. Technical report, 2008.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, J. Turner, and S. Shenker. Openflow: Enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, 2008.

[12] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *Symposium On Architecture For Networking And Communications Systems*, 2008.

[13] S. Ratnasamy, A. Ermolinskiy, and S. Shenker. Revisiting IP multicast. In *Proceedings of ACM SIGCOMM'06*, Pisa, Italy, Sept. 2006.

[14] A. C. Snoeren. Hash-based IP traceback. In *SIGCOMM '01*, pages 3–14, New York, NY, USA, 2001. ACM.

[15] T. Wolf. A credential-based data path architecture for assurable global networking. In *Proc. of IEEE MILCOM*, Orlando, FL, Oct 2007.

[16] A. Zahemszky, A. Csaszar, P. Nikander, and C. Esteve. Exploring the pubsub routing/forwarding space. In *International Workshop on the Network of the Future*, 2009.